RMIT UNIVERSITY

Advanced Programming Techniques
COSC1076 | Semester 2 2020
Assignment 1 (v1.0) | Robot Wall Following

Assessment Type	Individual Assessment. Clarifications/updates may be made via announcements/relevant discussion forums.
Due Date	11.59pm, Sunday 23 August 2020 (Before Week 6)
Silence Policy	From 5.00pm, Friday 21 August 2020 (Week 5)
Weight	15% of the final course mark
Submission	Online via Canvas. Submission instructions are provided on Canvas.

Change Log

1.1
- Fixed Milestone 3 sample output
- Fixed formatting of pseudocode

1.0
- Initial Release

1 Overview

In this assignment you will implement a **wall following** algorithm to help a simulated robot find the exit of simple 2D mazes.

In this assignment you will:

- Practice the programming skills such as:
  - Pointers
  - Dynamic Memory Management
  - Arrays
- Implement a medium size C++ program using predefined classes
- Use a prescribed set of C++11/14 language features

This assignment is marked on three criteria, as given on the Marking Rubric on Canvas:

- **Milestone 1:** Writing Tests
- **Milestone 2-4:** Implementation of the Wall Follower
- **Style & Code Description:** Producing well formatted, and well documented code.

Figure 1: An example robot

1.1 Relevant Lecture/Lab Material

To complete this assignment, you will require skills and knowledge from workshop and lab material for Weeks 2 to 4 (inclusive). You may find that you will be unable to complete some of the activities until you have completed the relevant lab work. However, you will be able to commence work on some sections. Thus, do the work you can initially, and continue to build in new features as you learn the relevant skills.

1.2 Learning Outcomes

This assessment relates to all of the learning outcomes of the course which are:

- Analyse and Solve computing problems; Design and Develop suitable algorithmic solutions using software concepts and skills both (a) introduced in this course, and (b) taught in pre-requisite courses; Implement and Code the algorithmic solutions in the C++ programming language.
- Discuss and Analyse software design and development strategies; Make and Justify choices in software design and development; Explore underpinning concepts as related to both theoretical and practical applications of software design and development using advanced programming techniques.
- Discuss, Analyse, and Use appropriate strategies to develop error-free software including static code analysis, modern debugging skills and practices, and C++ debugging tools.
- Implement small to medium software programs of varying complexity; Demonstrate and Adhere to good programming style, and modern standards and practices; Appropriately Use typical features of the C++ language include basic language constructs, abstract data types, encapsulation and polymorphism, dynamic memory management, dynamic data structures, file management, and managing large projects containing multiple source files; Adhere to the C++14 ISO language features.
- Demonstrate and Adhere to the standards and practice of Professionalism and Ethics, such as described in the ACS Core Body of Knowledge (CBOK) for ICT Professionals.

2 Background

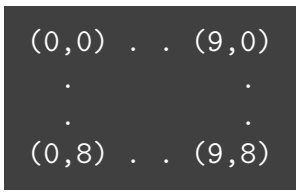
One challenge in robotics is *navigation*. This is the process of a robot moving between two different locations within some environment. A very simple algorithm that lets a robot drive about a very simple 2D maze is called **Wall Following**. In this assignment you will implement a simple wall following algorithm for a robot moving about a simple 2D maze. We will represent a simple 2D maze as a grid of ASCII characters. For example:

```
==========
=>.......=
========.=
=......=.=
==.=.=...=
=..=..=.==
=.===.=..=
=..==G==.=
==========
```

Aspects of the maze are represented by different symbols:

Symbol	Meaning
= (equal)	Wall or Obstacle within the maze. The robot cannot pass obstacles
. (dot)	Empty/Open Space.
>	The robot (see below for more information)
G	The goal point that the robot is trying to reach.

Each location in the maze is indexed by a cartesian `(x,y)` co-ordinate. The top-left corner of the maze is always the co-ordinate `(0,0)`, the x-coordinate increases right-wards, and the y-coordinate increases down-wards. For the above maze, the four corners have the following co-ordinates:



Mazes can be designed in different ways. For this assignment we will use mazes where:

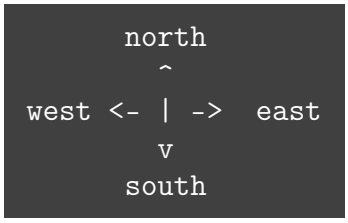
1. There is one goal (ending) point.
2. The robot's starting point is denoted by an "orientation" symbol (described below)
3. The maze is *always* surrounded by walls.
4. The maze *only* contains *junctions* and *corridors*. That is, you can't have "open" space, loops or islands.

The robot is represented by two properties:

- Its `(x,y)` co-ordinate within the maze
- The direction that it is "facing" , that is, its *orientation*. The orientation is represented by four characters:

Symbol	Key-code	Meaning
`<`	less-than symbol	Robot is facing `west`
`>`	greater-than symbol	Robot is facing `east`
`^`	hat/exponent symbol (`shift-6`)	Robot is facing `north`
`v`	lowercase letter v	Robot is facing `south`

To describe the robot's **orientation**, we use the 4 cardinal directions:



Thus, the **position** of the robot within the map is described as a 3-tuple of `(x,y,orientation)`. For example, in the above maze the robot starts at position `(1,1,east)`.

The robot can move about the maze using one of three actions:

1. Rotate clockwise 90°, changing it's *orientation*
2. Rotate counter-clockwise 90°, changing it's *orientation*
3. Move one space in the direction it is facing, , changing it's `x,y` co-ordinate

In this assignment you will implement a **simplified right-wall following algorithm** to enable the robot navigate from its starting position to the goal, which is described below.

! While there are many ways to navigate a maze, **you must implement this algorithm**.

Pseudocode for the Right Wall Following

```
Let M be the maze
Let T be the trail of the robot's positions
Let G be goal location for the robot to get reach
Let r be the current position (x, y, orientation) of the robot
repeat
    if There is a wall to the right-side of the robot then
        if The space in front of the robot is empty then
            Robot moves one space forward, and update r accordingly
            Add the updated r to the end of the trail T
        else if The space in front of the robot is not empty then
            Robot turns left (i.e. counter-clockwise), and update r accordingly
            Add the updated r to the end of the trail T
        end
    else
        Robot turns right (i.e. clockwise), and update r accordingly
        Add the updated r to the end of the trail T
        Robot moves one space forward right, and update r accordingly
        Add the updated r to the end of the trail T
    end
until The robot reaches the goal, that is, r == G (igorning the robot's orientation)
```

2.1 Wall Following Algorithm

The algorithm (given above) simulates the robot moving about the maze by imagining it places a "hand" on the wall to its **right**, and the moves by following the wall, never removing its "hand". This algorithm is given in *pseudocode*. This is a **right-wall follow**, and to keep things simple, we will stick to just following the right-wall in this assignment. Provided that the maze meets the requirements listed above, the robot is guaranteed to reach the goal. This might not be the fastest way to reach the goal, but it will get there eventually. On Canvas, you will find a video about this algorithm.

2.2 Outputting the path the robot took

Running the algorithm to get the robot to the goal is one step for this assignment. The second step is showing the path of the robot in navigating from where it started until it reached the goal. To do this, the algorithm tracks a history or *trail* of the robot's positions as it makes its way to the goal. For example, using the maze from the background section, the robot's path is below:

```
==========
=>>>>>>>v=
========v=
=>>>v<<=v=
==^=v=^<<=
=>^=>v=.==
=^===v=..=
=^<==G==.=
==========
```

Take careful note of the following. A robot may be at the same x, y position in the maze multiple times, but with different orientations, such as when:

- Is moves into a space and turn rotates (turns)
- Revisiting a location after backtracking

When showing the output of the path the robot travelled, the output must show the orientation of the robot for **the last time** that the robot was at that x, y location. For example, in the above maze, when the robot reached the dead-end at location `(2,7)`, by following the right wall, the robot then turned around the made its

way out of the dead-end. Therefore, the orientation arrows show the robot's movements as it came *out of the dead-end*. Further, at location `(4,3)`, the orientation is, `v` (`south`), since after coming our of the dead end, the robot followed the right-wall downward.

3 Assessment Details

The task for this assignment is to write a full C++ program that:

1. Reads in a 20x20 maze from standard-input (`std::cin`).
2. Finds the robot's starting position within the maze.
3. Executes the right-wall following algorithm (Section 2.1) until the robot reaches the goal
4. Prints out maze to standard output (`std::cout`), filled-in with the path the robot travelled.
5. (Milestone 3 only) Prints out walking/navigation directions.

You may assume that the maze is ***always*** a fixed size of 20x20, except for Milestone 4.

This assignment has four Milestones. To receive a PA/CR grade, you only need to complete Milestones 1 & 2. To receive higher grades, you will need to complete Milestones 3 & 4.

> ! Take careful note of the Marking Rubric on Canvas. Milestones should be completed in sequence.

3.1 Milestone 1: Writing Tests

Before starting out on implementation, it is good practice to write some tests. We are going to use I/O-blackbox testing. That is, we will give our program a maze to solve (as Input), and then test that our program's Output is what we expect it to be.

A test consists of two text-files:

1. `<testname>.maze` - The *input* maze for the program to solve
2. `<testname>.out` - The *expected output* which is the solution to the maze.

A test *passes* if the output of your program *matches* the expected output. Section 4.4 explains how to run your tests using the `diff` tool.

You should aim to write a minimum of **four tests**. We will mark your tests based on how suitable they are for testing that your program is 100% correct. Just having four tests is not enough for full marks.

3.2 Milestone 2: Wall Following

It is important to have a good *design* for our programs and use suitable *data structures* and *classes*. In Assignment 1, you will implement our design[1]. You will implement 3 classes:

- `Position` class - to represent a position (x, y, *orientation*) of the robot.
- `Trail` class - to record the trail of positions as the robot navigates the maze.
- `WallFollower` class - that executes the wall following algorithm.
- The `main` file that uses these classes, and does any reading/writing to standard input/output.

You are given these classes in the starter code. You may add any of your own code, but you **must not modify** the definitions of the provided class methods and fields.

3.2.1 `Position` Class

The `Position` class represents a position of the robot. It is a tuple `(x,y,orientation)`, which is the `x-y` location of the robot, and the direction (orientation) that the robot is "facing." It only contains accessors methods for this information.

[1] This won't be the case for Assignment 2, where you will have to make these decisions for yourself.

```
// Constructor/Destructor
Position(int x, int y, Orientation orientation);
~Position();

// x-co-ordinate of the robot
int getX();

// y-co-ordinate of the robot
int getY();

// Orientation the robot is facing
Orientation getOrientation();
```

Notice that `Orientation` is an enumeration:

```
enum Orientation {
   ORIEN_NORTH,
   ORIEN_EAST,
   ORIEN_SOUTH,
   ORIEN_WEST,
};
```

3.2.2 `Trail` Class

The `Trail` class stores the history of positions of the robot as it conducted the wall-following. It stores an *array* of `Position` objects. Since it's an array we also need to track the number of position objects in the trail.

> ! You *must* implement the `Trail` class using an *array*.

```
// Trail of position objects
// You may assume a fixed size for M1 & M2
Position* trail[TRAIL_ARRAY_MAX_SIZE];

// Number of objects currently in the trail
int length;
```

The constant `TRAIL_ARRAY_MAX_SIZE` is the maximum number of objects that can be in a trail. This constant is given in the `Types.h` header file.

```
#define MAZE_DIM                20
#define TRAIL_ARRAY_MAX_SIZE    (4 * MAZE_DIM * MAZE_DIM)
```

The `Trail` class has the following methods:

```
// Constructor/Destructor.
Trail();
~Trail();

// Copy constructor - create a DEEP COPY of the Trail
Trail(Trail& other);

// Number of elements in the Trail
int size();

// Get a pointer to the i-th trail element in the array
Position* getPosition(int i);

// Add a COPY of the Position object to the BACK of the trail.
void addCopy(Position* position);
```

These methods let you add positions to the trail, and get a pointer to an existing position. Be aware, that the Trail class has full control over all position objects that are stored in the array. Thus, if position objects are removed from the array you must remember to "delete" the objects.

3.2.3 `WallFollower` Class

The `WallFollower` class executes the right-wall following algorithm by using the Trail and Position classes. It has two main components:

1. Execute the wall following algorithm
2. Get a **deep copy** of the full path that the robot took

```
// Constructor/Destructor
WallFollower();
~WallFollower();

// Execute the wall following algorithm for the given maze
void execute(Maze maze);

// Get a DEEP COPY of the path the robot travelled
Trail* getFullPath();
```

This uses a custom data type `Maze`, which is given in the `Types.h`. It is a 2D array of characters that represents a maze using the format in Section 2. It is a fixed size, because we assume the size of the maze is known.

```
// A 2D array to represent the maze
// REMEMBER: in a maze, the location (x,y) is found by maze[y][x]!
typedef char Maze[MAZE_DIM][MAZE_DIM];
```

It is very important to understand the `Maze` type. It is defined as a 2D array. If you recall from lectures/labs, a 2D array is indexed by *rows* then *columns*. So if you want to look-up a position `(x,y)` in the maze, you find this by `maze[y][x]`, that is, first you look-up the `y` value, *then* you look-up the `x` value.

The `execute` method is given a maze, and conducts the wall following algorithm in Section 2.1. Remember, that the initial position of the robot is recorded in the maze. Importantly, the `execute` method **must not** modify the maze it is given.

The trail of positions that the robot takes is stored in the private field:

```
// Trail of positions from the start to end
Trail* path;
```

The `getFullPath` method returns a **deep copy** of the trail of the robot's positions in executing the wall following. Be aware that this is a **deep copy** of the trail, so you need to return a new Trail object.

3.2.4 `main` file

The `main` file:

1. Reads in a maze from standard input
2. Executes the wall following algorithm
3. Gets the full navigation path
4. Outputs the maze (with the path) to standard output

The starter code gives you the outline of this program. It has two functions for you to implement that read in the maze and print out the solution.

```
// Read a maze from standard input.
void readMazeStdin(Maze maze);

// Print out a Maze to standard output.
void printMazeStdout(Maze maze, Trail* path);
```

Some hints for this section are:

- You can read one character from standard input by:

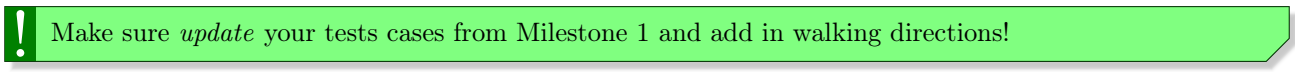
```
char c;
std::cin >> c;
```

- Remember that is *ignores all white space* including newlines!
- When printing the maze, you might find it easier to first update the maze with navigation path, and then print out the whole maze.

3.3 Milestone 3: Robot Directions

For Milestone 3, reconsider the output in Section 2.2. It contains a long backtrack section. It would be nice to generate the actual list of movements of the robot with the backtracking removed, so we get the most direct route from the starting position to the goal. These movements will be a series of cardinal directions,

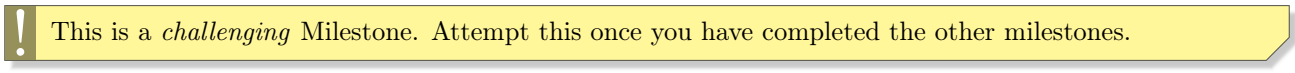
After your program shows the solution to the maze, your program should print out *in lowercase* the cardinal directions following the arrows generated in Milestone 2, but without any backtracking. Each direction is printed in order one-at-a-time, each on a new line (as below). *Hint:* The Milestone 2 algorithm should be very helpful in generating the correct list of directions.


Make sure *update* your tests cases from Milestone 1 and add in walking directions!

For example, if we use the maze and solution from Section 2, the walking directions would be:

```
east
east
east
east
east
east
east
south
south
south
west
west
north
west
west
south
south
east
south
south
```

3.4 Milestone 4: Dynamic Array Allocation


This is a *challenging* Milestone. Attempt this once you have completed the other milestones.

For Milestones 1 - 3, we assume that the maze is *always* of a fixed size (`20x20`). This means, that for the `Maze` data type and the `Trail` class we could define the size of the arrays before-hand.

For Milestone 4, you must modify your implementation to accommodate two significant changes:

- Use a Maze of *any rectangular* size.
- Dynamically resize the `path` field of the `Trail` as more elements are required in the array, rather than use a fixed size.

To do this, you will need to modify a number of aspects of your implementation to **dynamically** allocate memory for 1D and 2D arrays. You will need to consider the following modifications:

- Change the type of `Maze` to a generic 2D array:

```
typedef char** Maze;
```

  The `milestone4.h` file in the starter code has a sample method to help you dynamically allocate memory for a 2D array.
- Change the type of the field `path` in the `Trail` class to a generic 1D array of pointers:

```
Position** path;
```

- Create memory as necessary for the maze and trail.
- When reading in the maze, you will need to be able to spot newline characters. You can't do this if you follow the suggestion for Milestone 2. Instead you will need the `get` method of `std::cin`:

```
char c;
std::cin.get(c);
```

  See the C++ Reference documentation for more information.

3.5 Documentation, Style and Code Description

Making sure your code is 100% correct is very important. Making sure your code is understandable is equally important. Your code should follow the Course Style Guide, as given on Canvas (including not using any banned elements), and should be well documented with clear comments.

Finally, you need to provide a **short 1-paragraph description** (at the top of your `main` file) to:

- Describe (briefly) the approach you have taken in your implementation
- Describe (briefly) any issues you encountered

If you completed Milestones 3 or 4, this code description should include what you had to do for these milestones.

> ! You may only use C++ languages features and STL elements that are covered in class.

4 Getting Started

4.1 Starter Code

We have provided starter code to help you get underway. This includes files for the classes, the `Types.h` and `main` files, and the `milestone4.h` header for Milestone 4. Ignore this if unless you attempt this milestone.

To compile your program, you will need to use a command similar to the following:

```
g++ -Wall -Werror -std=c++14 -O -o assign1 Position.cpp Trail.cpp WallFollower.cpp main.cpp
```

4.2 Suggestions for starting Milestone 1

The starter code also contains a folder with one sample test case for Milestone 2, and a separate expected output file if you complete Milestone 3. This should give you an idea of how your tests should be formatted.

4.3 Suggestions for starting Milestone 2

Part of the learning process of the skill of programming is devising how to solve problems. In this assignment, the problem solving is turning an *algorithm* and *pseudocode* into a *complete functioning program*.

This process involves completing small tasks one-at-a-time. We recommend the sequence of tasks:

1. In the `main` file, read in a maze from standard input and print out this maze (unmodified)
2. Implement the `Position` class
3. Implement the `Trail` class
4. Implement the `WallFollower` class
5. Update the `main` file to use the `WallFollower`

Testing is also an important part of this process. The tests you need to write for Milestone 1 test *your whole program*. This has a problem, because this means you have to write the whole program first. However, you can write small programs to ***test your program as you go***. The `main` file in the starter code has a couple of examples to help you test that your `Position` and `Trail` class as you develop them. Of course, once you finish the assignment, you can delete this testing code. This **lets you test small parts of your program as you go** rather than waiting until the end and just hoping the whole thing works.

4.4 Running Milestone 1 Tests

As a reminder, you can run a test as below. Recall that is uses the `diff` program to compare the *actual* and *expected* output of your program.

```
./assign1 <testname.maze >actual.out
    diff actual.out testname.out
```

5 Submission

Follow the detailed instructions **on Canvas** to complete your submission for Assignment 1.

Assessment declaration: When you submit work electronically, you agree to the assessment declaration.

5.1 Silence Period

A silence policy will take effect from **5.00pm, Friday 21 August 2020 (Week 5)**. This means no questions about this assignment will be answered, whether they are asked on the discussion board, by email, or in person. Make sure you ask your questions with plenty of time for them to be answered.

5.2 Late Submissions & Extensions

A penalty of 10% per day is applied to late submissions up to 5 days, after which you will lose ALL the assignment marks. Extensions will be given only in exceptional cases; refer to Special Consideration process.

Special Considerations given after grades and/or solutions have been released *will automatically result in an equivalent assessment in the form of a test*, assessing the same knowledge and skills of the assignment.

6 Marking guidelines

The marks are divided into three categories:

- Tests: 2/15 (15%)
- Software Implementation: 9/15 (60%)
- Code Style, Documentation & Code Description: 4/15 (25%)

The detailed breakdown of this marking guidelines is provided on the rubric linked on Canvas.

Please take note that the rubric is structured with with three "brackets":

- If you do a good job on Milestone 1 & 2, then your final mark will be a CR. This will mean you have a CR in all three rubric categories
- If you do a good job for Milestone 3, then you mark will be a DI, getting a DI in all rubric categories
- If you do a good job for Milestone 4, your mark will be a HD.

The purpose of this is for you to focus on successfully completing each Milestone *one-at-a-time*. You will also notice there are not many marks for "trying" or just "getting started". This is because this is an *advanced* course. You need to make *significant* progress on solving the task in this assignment before you get any marks.

7 Academic integrity and plagiarism (standard warning)

CLO 6 for this course is: *Demonstrate and Adhere to the standards and practice of Professionalism and Ethics, such as described in the ACS Core Body of Knowledge (CBOK) for ICT Professionals.*

Academic integrity is about honest presentation of your academic work. It means acknowledging the work of others while developing your own insights, knowledge and ideas. You should take extreme care that you have:

- Acknowledged words, data, diagrams, models, frameworks and/or ideas of others you have quoted (i.e. directly copied), summarised, paraphrased, discussed or mentioned in your assessment through the appropriate referencing methods
- Provided a reference list of the publication details so your reader can locate the source if necessary. This includes material taken from Internet sites. If you do not acknowledge the sources of your material, you may be accused of plagiarism because you have passed off the work and ideas of another person without appropriate referencing, as if they were your own.

RMIT University treats plagiarism as a very serious offence constituting misconduct. Plagiarism covers a variety of inappropriate behaviours, including:

- Failure to properly document a source
- Copyright material from the internet or databases
- Collusion between students

For further information on our policies and procedures, please refer to the RMIT Academic Integrity Website.

The penalty for plagiarised assignments include zero marks for that assignment, or failure for this course. Please keep in mind that RMIT University uses plagiarism detection software.